Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Danilo Šormaz

Primjena graf algoritama za pronalaženje optimalne rute na mapama

Završni rad

Osijek, 2016.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Danilo Šormaz

Primjena graf algoritama za pronalaženje optimalne rute na mapama

Završni rad

Mentor: izv.prof.dr. sc. Domagoj Matijević
Sumentor: dr. sc. Domagoj Ševerdija

Osijek, 2016.

Sadržaj

Sažetak

Glavna tema ovog rada su graf algoritmi za pronalaženje najkraće rute na mapama. U prvom poglavlju definirat ćemo graf u matematičkom smislu, spomenuti i objasniti vrste grafova te dati par primjera različitih tipova grafova.U drugom dijelu poglavlja predstavit ćemo načine reprezentacije grafa u memoriji računala.

U drugom poglavlju definirat ćemo šetnju kroz graf i putove u grafu. Nakon toga ćemo pojasniti obilazak grafova pomoću pretraživanja grafa po širini i dubini.Objasnit ćemo problem najkraćeg puta, raspisati Dijsktrin algoritam i na osnovu njega izvesti tehnike ubrzavanja kao što su dvosmjerno pretraživanje, A* i ALT.

Ključne riječi

Grafovi, problem nakraćeg puta, planiranje rute, pretraživanje grafa po širini i dubini.

Summary

The main topic of this thesis are shortest path graph algorithms. In the first part of the paper we shall define a graph in mathematical sense, mention and explain types of graphs and give some examples of different graphs. In the second part of the first chapter we'll present how to store or represent graph in comupter memory.

In the second part we shalll define walk and paths in graph. After that Breadth-First Search and Depth-First Search is presented. We'll explain shortest path problem, write out Dijkstra's algorithm and presented speed-up techniques such as bidirectional search, A* and ALT.

Keywords

Graphs, shortest path problem, route planning, Breadth-First Search, Depth-First Search.

Uvod

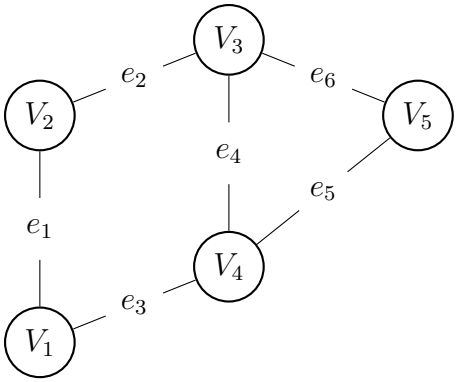
Računanje najbolje moguće rute u cestovnoj mreži od zadanog ishodišta do određenog cilja je svakodnevni problem. Mnogo ljudi se redovno susreće s ovim problemom prilikom planiranja putovanja. Skupljanje informacija o mapama je do sada prilično napredovalo i cestovne mreže su se razvile tako da sadrže veliki broj raskrižja. Zbog toga, korištenje jednostavnih pristupa rješavanju najkraćeg puta nije preporučljivo, što zbog vrlo sporog obavljanja upita, što zbog neefikasne memorijske iskoristivosti. S druge strane korištenje agresivne heuristike može dovesti do pogrešnih rezultata. Zbog toga je potrebno izbalansirati prethodno navedene probleme kako bi rješenje problema bilo što optimalnije i kako bi klijent bio zadovoljan uslugom. U današnje vrijeme, redovno se razvijaju sve točnije i efikasnije tehnike pronalaženja najkraćeg puta između dvije točke na mapi.

1 Teorija grafova

1.1 Matematička definicija grafa

Definicija 1.1. *Graf G je uređeni par $G = (V, E)$, gdje je $V(G) = V \neq \emptyset$ skup vrhova (engl. vertex), a $E(G) = E \subseteq \binom{V}{2}$ skup bridova (engl. edge), gdje $\binom{V}{2}$ predstavlja sve dvočlane podskupove od V. Svaki brid $e \in E$ spaja dva vrha $u, v \in V$ koji se zovu krajevi od e, te kažemo da je brid e incidentan s vrhovima u, v.*

Primjer 1.1. *Graf G=(V,E) sa vrhovima $V = \{V_1, V_2, ..., V_5\}$, i bridovima $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. Svakom bridu je pridružen 2-člani multiskup vrhova :*
$e_1 = \{V_1, V_2\}, e_2 = \{V_2, V_3\}, e_3 = \{V_1, V_4\}, e_4 = \{V_3, V_4\}, e_5 = \{V_4, V_5\}, e_6 = \{V_3, V_5\}$



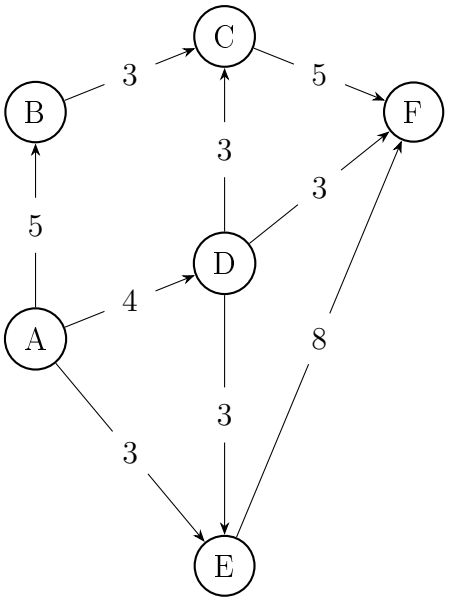
Slika 1: Graf $G = (V, E)$

U prikazanom Primjeru 1.1. bridovi grafa su neusmjereni tj. po svakom bridu se može "ići u oba smjera". Prema tome iz vrha V_1 se moglo doći u vrh V_2 i obratno.

Definicija 1.2. *Digraf ili usmjereni graf D je uređena trojka $D = (V, E, \psi)$, gdje je $V = V(D) \neq \emptyset$ skup čije elemente nazivamo vrhovima, $E = E(D)$ skup disjunktan s V čije elemente nazivamo bridovima i ψ funkcija koja svakom bridu $e \in E$ pridružuje uređeni par (u, v). Kažemo da je u početak, a v kraj brida e, da je smjer ili orjentacija brida a od u prema v i koristimo oznake $a = (u, v)$. Digraf skraćeno označavamo s $D(V, E)$ ili samo D.*

U praksi se javlja velika potreba prikazivanja različitih sustava pomoću digrafova (npr. regulacija prometa u nekom gradu, protok tekućine u nekom sustavu, transport robe...). U daljnjem tekstu pod pojmom grafa podrazumijevat ćemo da je svejedno o kojem tipu grafa misilimo, ukoliko nije eksplicitno naznačeno da je on ili usmjeren ili neusmjeren.

Definicija 1.3. *[2] Težinski graf $G^\alpha(V, E)$ je (usmjereni) graf na čijim bridovima je definirana težinska funkcija $\alpha\colon E \to \mathbb{R}$ pri čemu broj $\alpha(e)$ zovemo težinom brida $e \in E$.*



Slika 2: Usmjereni težinski graf

Definicija 1.4. *[3] Stupanj vrha v u grafu $G = (V, E)$ je broj bridova grafa G incidentnih s v.Oznaka za stupanj vrh v je $deg(v)$.*

Lema 1.1. *(O rukovanju) U svakom neusmjerenom grafu $G = (V, E)$ je zbroj stupnjeva svih vrhova paran, tj. vrijedi*

$$\sum_{v \in G} deg(v) = 2|E| \tag{1}$$

Dokaz. Tvrdnju dokazujemo prebrajanjem svih "incidencija" grafa tj. skupa $\{(v, e) : v \in V(G), e \in E(G), v \in e\}$ na dva načina.Krenemo li od vrhova, za svaki pojedini vrh takvih incidencija ima točno koliko je stupanj dotičnog vrha. Krenemo li od bridova, vidimo da svaki brid ima dva kraja, tj. da je dvočlani podskup, pa sveukupno incidencija ima $2|E(G)|$. Time smo dokazali ovu jednakost. Kako je desna strana jednakosti očevidno parna, budući je višeratnik broja 2, parna mora biti i lijeva strana što upravo dokazuje tvrdnju leme. □

1.2 Graf kao struktura podataka

Za algoritamsku obradu određenog grafa potreban nam je sustavan i praktičan način prikaza njegovih vrhova i bridova. Ovo je posebno bitno prilikom pohrane grafa u računalo jer različit izbor reprezentacije grafa zauzima različitu količinu ograničene memorije, a i vrijeme izvršavanja algoritma se razlikuje.
U daljnjem tekstu ćemo broj vrhova u grafu označavati s $|V|$, a broj bridova $|E|$. Dva standardna prikaza grafa $G = (V, E)$ je kao liste susjedstva i matrica susjedstva. Primjenjuju se i na usmjerenim i neusmjerenim grafovima.

Matrica susjedstva Svakom grafu s $|V|$ vrhova možemo pridružit matricu $A \in |V| \times |V|$ u čijem se i-tom retku i j-tom stupcu nalazi broj bridova koji spajaju i-ti i j-ti vrh. Matricu

s tim svojstvom nazivamo matrica susjedstva.

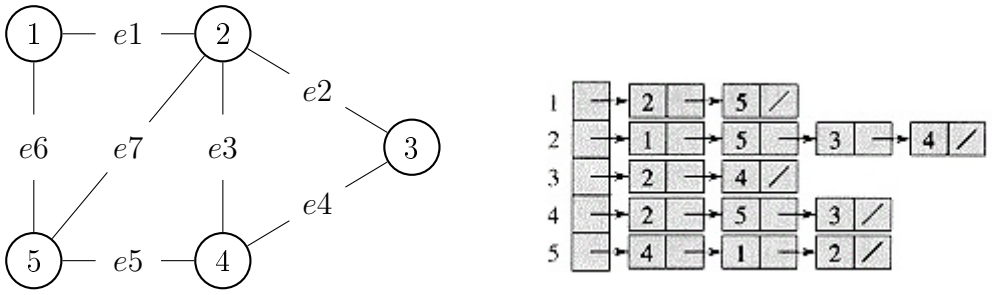
Primjer 1.2. *Prikaz tablice, odnosno matrice susjedstva neusmjerenog grafa iz Primjera 1.1.*

	V_1	V_2	V_3	V_4	V_5
V_1	*0*	*1*	*0*	*1*	*0*
V_2	*1*	*0*	*1*	*0*	*0*
V_3	*0*	*1*	*0*	*1*	*1*
V_4	*1*	*0*	*1*	*0*	*1*
V_5	*0*	*0*	*1*	*1*	*0*

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Kod jednostavnih(nema bridova čiji se krajevi podudaraju tj. petlji, i višestrukih bridova) neusmjerenih težinskih grafova na mjesto *(i,j)* dolazi težina brida. Ukoliko je graf neusmjeren tada je $a_{ij} = a_{ji}$ odnosno matrica je simetrična. Za vrhove i i j koji nisu povezani pisat ćemo $a_{ij} = 0$ ili $a_{ij} = \infty$ u zavisnosti o datom problemu. U mnogim slučajevima je u redu koristiti 0, ali ukoliko graf ima i negativne bridove tada pišemo ∞. Za graf bez petlje vrijedi $i = j \Rightarrow a_{ij} = 0$ ili $a_{ij} = \infty$.
Ovaj način reprezentacije grafa je relativno jednostavno implementirati, ali zauzima veću količinu memorije od npr. liste bridova(praćenje liste parova vrhova koji predstavljaju krajeve bridova). Matrica susjedstva grafa $G = (V, E)$ zauzima $\Theta(|V|^2)$ memorije, nezavisno o broju bridova u grafu.

Liste susjedstva Liste susjedstva pružaju mnogo bolji način reprezentacije za rijetke grafove tj. grafove kojima je broj bridova mnogo manji od broja vrhova. Takvi grafovi bi u matrici susjedstva imali veliki broj nula što onda ne bi bio najefikasniji način pohrane u računalo. Kako bi to uspješno izbjegli koristimo liste susjedstva koje zapravo prestavljaju polje *Adj* od $|V|$ elemenata, gdje je i-tom elementu pridružena lista susjednih vrhova od vrha v_i.To bi značilo da za svaki vrh $u \in V$ lista $Adj[u]$ sadrži sve vrhove v takve da postoji brid $(u, v) \in E$.
Ukoliko je graf G neusmjeren, zbroj dužina svih susjednih lista je $2|E|$, jer je brid oblika *(u,v)* takav da se vrh v pojavljuje u listi $Adj[u]$ i obratno(Lema 1.1). Ako je G usmjeren graf tada je gornja granica broja njegovih bridova također $2|E|$. Količina memorije koju zahtjevaju liste susjedstva je zadovoljavajuća i iznosi $\Theta(V + E)$ što je optimalnije za razliku od matrice susjedstva. Nedostatak ove reprezentacije je dosta spora provjera postojanosti određenog brida u grafu u odnosu na matrice susjedstva.



Slika 3: Reprezentacija grafa listom susjedstva [1]

2 Najkraći put

2.1 Šetnje i putovi

Definicija 2.1. *Šetnja u grafu $G = (V, E)$ je niz $W := v_0e_1v_1e_2...e_kv_k$ čiji članovi su naizmjenično vrhovi v_i i bridovi e_i ,tako da su krajevi od e_i vrhovi v_{i-1} i v_i,i=1,...,k. Kažemo da je v_0 početak ,a v_k kraj šetnje W, tj. da je (v_0, v_k) šetnja.*

Definicija 2.2. *Ako su svi bridovi $e_1, ..., e_k$ šetnje W međusobno različiti, onda se W zove staza.*

Definicija 2.3. *Ako su na stazi W svi vrhovi $v_1, ..., v_k$ međusobno različiti, onda se šetnja zove put.*

Primjer 2.1. *Primjer šetnje, staze i puta grafa iz Primjera 1.1.*
Šetnja: $V_1e_1V_2e_2V_3e_4V_4e_5V_5e_6V_3e_2V_2$.
Staza: $V_1e_1V_2e_2V_3e_4V_4e_3V_1$.
Put: $V_1e_1V_2e_2V_3e_6V_5$.

2.2 Obilazak grafova

Obilazak grafa je prolazak njegovim bridovima kako bi se obišli svi njegovi vrhovi. Prolaskom kroz graf mnogo toga se saznaje o samoj strukturi grafa. Možemo saznati da li je graf povezan i od kojih se komponenti sastoji ako nije, ima li u njemu ciklusa itd. Algoritmi za obilazak grafa predstavljaju temelj drugih vrlo bitnih algoritama u teoriji grafova. Dva najčešća algoritma za pretraživanje grafa su pretraživanje u širinu (engl. Breadth-First Search) i pretraživanje u dubinu (engl. Depth-First Search).

Pretraživanje u širinu(BFS) Pretraživanje u širinu je jedan od najjednostavnijih algoritama za pretraživanje grafa. Za zadani graf $G = (V, E)$ i određeni početni vrh s, BFS sistematski pretražuje bridove grafa G kako bi pronašao svaki vrh koji se može posjetiti počevši od vrha s. Ideja je da se iz vrha s prvo posjete svi susjedni vrhovi, a nakon toga svi "susjedi susjeda" itd. sve dok se ne posjete svi vrhovi dostupni iz s. BFS računa i udaljenost (najmanji broj bridova) od s do svih dostupnih vrhova.

Ime je dobio po tome što se granica između otkrivenih i neotkrivenih vrhova "širi" kroz graf tj. algoritam prvo otkriva vrhove na udaljenosti k od s, a tek onda one na udaljenosti $k+1$.

Kako bi pratio napredak pretraživanja, algoritam boji vrhove bijelom, sivom ili crnom bojom. Svi vrhovi su na početku bijele boje, ali kasnije, u toku izvršavanja algoritma, mogu postati sivi, a potom i crni. Vrh je otkriven kada se prvi put, prilikom pretraživanja susretnemo s njim, i tada taj vrh više nije bijele boje. I sivi i crni vrhovi su otkriveni, ali BFS pravi razliku između njih. Ako je $(u,v) \in E$ i vrh u crne boje, tada je vrh v ili sive ili crne boje, tj. svi vrhovi susjedni crnom vrhu su otkriveni. To znači da je vrh u crne boje ukoliko su svi susjedi od u otkriveni, a sive je boje ukoliko je vrh u otkriven, ali svi njegovi susjedi nisu. Prema tome sivi vrhovi mogu imati i bijele susjede, te oni (sivi) predstavljaju granicu između otkrivenih i neotkrivenih vrhova.

BFS konstruira stablo(BF) koje u početku sadrži samo korijen odnosno ishodišni vrh s. Svaki put kada BFS otkrije bijeli vrh v, prilikom pretrage liste susjedstva već otkrivenog vrha u, vrh v i brid (u,v) će biti dodani stablu. Tada je u prethodnik ili roditelj vrha v u BF stablu.

Pretraživanje u širinu za reprezentaciju grafa koristi liste susjedstva. U algoritmu ćemo s $u.boja$ označiti boju vrha $u \in V$, prethodnika od u s $u.\pi$. Ukoliko u nema prethodnika ($u = s$ ili u nije otrkiven), tada je $u.\pi = NIL$. S $u.d$ ćemo označiti najmanji broj bridova od s do u. Algoritam koristi strukturu podataka red (engl. queue, first in, first out) uz koju vežemo dvije operacije: možemo staviti nešto na kraj reda i uzeti nešto s početka reda tj. operacije Enqueue i Dequeue redom. U redu će uvijek biti samo sivi vrhovi; uzet ćemo jedan sivi vrh s početka reda, a sve njegove bijele susjede staviti na kraj reda pri čemu oni postaju sivi. Izbačeni sivi vrh iz reda postat će crni.

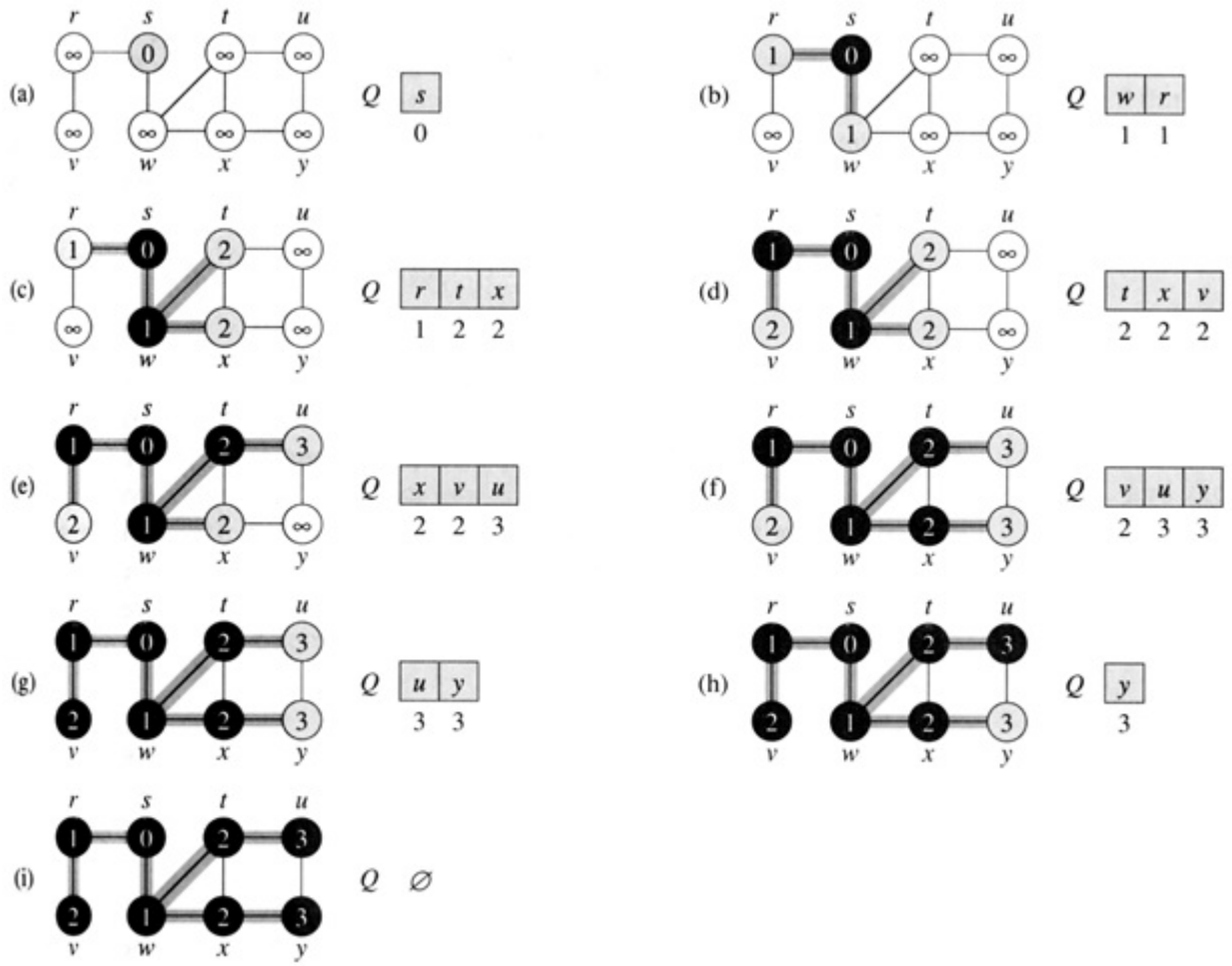
Algoritam pretraživanja u širinu(BFS)

BFS(G,s)

1. *za svaki vrh* $u \in G.V \setminus \{s\}$ *radi sljedeće:*
2. $u.boja$=*bijela*
3. $u.d = \infty$
4. $u.\pi = NIL$
5. $s.boja$=*siva*
6. $s.d$=*0*
7. $s.\pi = NIL$
8. $Q = \emptyset$
9. *ENQUEUE(Q,s)* (stavi s na kraj reda)
10. *dok je red Q neprazan radi sljedeće:*
11. u=*DEQUEUE(Q)* (uzmi vrh u s početka reda)
12. *za svaki* $v \in G.Adj[u]$ *radi sljedeće:*
13. *ako je* $v.boja$== *bijela tada:*
14. $v.boja$=*siva*
15. $v.d$=$u.d$+*1*

16. $v.\pi = u$
17. *ENQUEUE(Q,v)* (stavi v na kraj reda)
18. *u.boja=crna*

BFS radi na sljedeći način. Koraci od 1 do 4, sa iznimkom za vrh s, bojaju sve vrhove u bijelu boju, postavljaju sve udaljenosti na ∞ i za svaki vrh je prethodnik odnosno roditelj NIL. Posebno koraci od 5 do 7 bojaju vrh s u sivu boju, postavljaju udaljenost ishodišnog vrha na 0 i prethodnik na NIL. Koraci 8 i 9 inicijaliziraju Q kao red koji sadrži samo vrh s. Petlja u koraku od 10 do 18 se izvršava sve dok ima sivih vrhova, odnosno vrhova koji su otkriveni, ali čije susjedne liste još nisu pregledane. Algoritam uzima redom sive vrhove s početka reda (korak 11), pregledava vrhove u njihovim listama susjedstva (korak 12) i ako su bijeli otkriva ih, boji u sivo, postavlja im udaljenost $v.d$ na $u.d + 1$ i stavlja ih u red. Konačno, kad su svi vrhovi u listi susjedstva pregledani, vrh postaje crn.



Slika 4: Primjer pretraživanja u širinu neusmjerenog grafa [1]

Pretraživanje u dubinu(DFS) Ovaj algoritam također sistematično pronalazi sve vrhove, ali na nešto drukčiji način od pretraživanja u širinu. Umjesto da se širi u grafu, on uvijek nastoji ići "dublje" u graf, tj. za svaki susjedni vrh v vrha u pretraživanje u dubinu se rekurzivno poziva i provjerava susjede vrha v prije nego se vrati i provjeri preostale susjede

vrha u.

Slično kao kod BFS-a, i DFS boji vrhove kako bi pratilo napredak. Svaki vrh je na početku bijele boje, kad je otkriven postaje siv, a crn postaje kad su mu svi vrhovi u listi susjedstva otkriveni i provjereni. Za reprezentaciju grafa u algoritmu koristimo listu susjedstva kao i kod BFS-a.

ALGORITAM PRETRAŽIVANJA U DUBINU(DFS)

```
DFS(G)
1. za svaki vrh u ∈ G.V radi sljedeće:
2.         u.boja=bijela
3.         u.π = NIL
4. vrijeme=0
5. za svaki vrh u ∈ G.V radi sljedeće:
6.         ako je u.boja==bijela tada:
7.                 Posjeti(G,u)
```

```
Posjeti(G,u)
1. vrijeme=vrijeme+1
2. u.d=vrijeme
3. u.boja=siva
4. za svaki u ∈ G.Adj[u]
5.         ako je v.boja==bijela tada:
6.                 v.π = u
7.                 Posjeti(G,v)
8. u.boja=crna
9. vrijeme=vrijeme+1
10. u.f=vrijeme
```

Pretraživanje u dubinu svaki vrh "vremenski označava". Svaki vrh v ima dvije "vremenske oznake" : prva $v.d$ bilježi kada je vrh v prvi put otkriven (obojan u sivo), a druga $v.f$ bilježi kada pretraživanje završi pretraživanjem liste susjedstva vrha v (obojan u crno). U algoritmu varijabla *vrijeme* je globalna varijabla koju smo koristili za "vremenske oznake". DFS radi na sljedeći način: koraci od 1 do 3 obojaju svaki vrh u bijelo i inicijaliziraju njihove prethodnike na NIL. Korak 4 postavlja brojač *vrijeme* na 0. Koraci od 5 do 7 provjeravaju svaki vrh u V i kada je taj vrh bijele boje tada se poziva rekurzivna procedura *Posjeti*. Kada se procedura izvrši, svakom vrhu u će biti dodijeljeno "vrijeme otkrivanj"' $u.d$ i "vrijeme završetka" $u.f$.

U svakom pozivu procedure $Posjeti(G, u)$, vrh u je obojan bijelom bojom. Korak 1 povećava varijablu *vrijeme* za 1, korak 2 bilježi novu vrijednost varijable *vrijeme* kao "vrijeme otkrivanja" $u.d$, a korak 3 boja vrh u sivom bojom. Korak 4 provjerava svaki vrh v susjedan vrhu u i rekurzivno posjećuje v ukoliko je bijele boje. Na kraju, kada svi dostupni vrhovi iz vrha u budu posjećeni, u se oboja u crno (svi susjedni vrhovi otkriveni i završeni), *vrijeme* se poveća za 1 i spremi u "vrijeme završetka" $u.f$. "Vremenske oznake" nam daju neke dodatne informacije o grafu.

2.3 Problem najkraćeg puta

Zamislimo sljedeći problem: želimo pronaći najkraći cestovni put od Osijeka do Zagreba. Na raspolaganju nam je mapa Republike Hrvatske na kojoj je označena svaka duljina između dva susjedna raskrižja. Kako ćemo pronaći najkraći put? Jedan od načina bi bio kada bi pronašli sve putove od Osijeka do Zagreba, zbrojili duljinu svakog puta i u obzir uzeli minimalni odnosno najkraći put. Međutim lako je za primjetiti da bi tada morali ispitati ogroman broj putova od kojih većinu nema smisla provjeravati. Pokušat ćemo predočiti optimalnije načine pronalska najkraćeg puta .

Cestovna mreža se može prikazati u obliku grafa tj. kao skup vrhova V (raskrižja) i bridova E (dijelovi ceste) gdje svaki brid povezuje dva vrha. Svakom bridu je dodijeljena težina tj. duljina ceste ili npr. procjena vremena potrebnog za prolazak tim dijelom ceste. U teoriji grafova je računanje najkraćeg puta klasičan problem, no isto tako postoje varijacije (varijante) tog problema:

- "najkraći put od početnog do krajnjeg vrha " (engl. point to point) - pronalazi duljinu najkraćeg puta od danog izvora $s \in V$ do određenog cilja $t \in V$;
- "najkraći put od zadanog ishodišta" (engl. single source) - za dani izvor $s \in V$ računa duljinu najkraćeg puta do svih vrhova $v \in V$;
- "najkraći put između parova vrhova" (engl. many-to-many) - za dani skup vrhova $S, T \subseteq V$, računa duljinu najkraćeg puta za svaki par vrhova $(s,t) \in S \times T$;
- "najkraći put između svih parova vrhova " (enlg. all-pairs) - poseban slučaj ''many-to-many" varijante s $S := V := T$.

U ovom radu ćemo se prije svega baviti problemom pronalaženja najkraćeg puta od zadanog ishodišta do zadanog cilja odnosno od vrha do vrha. Prema tome put P u grafu G od vrha v_0 do vrha v_k je niz bridova $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Duljina, odnosno težina $w(P)$ puta P je zbroj težina bridova koje pripadaju putu P.
$w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
P* $=\langle s, ..., t\rangle$ je najkraći put ako ne postoji put P' od vrha s do vrha t takav da je $w(P') <$ w(P*). Preciznije težinu najkraćeg puta od u do v definiramo s:

$$\delta(u,v) = \begin{cases} min\ \{w(P)\text{: u} \xrightarrow{P} \text{v}\} & \text{,ako postoji put od } u \text{ do } v \\ \infty & \text{,inače} \end{cases}$$

Tada je najkraći put od vrha u do vrha v definiran kao bilo koji put P s težinom $w(P) = \delta(u,v)$

2.4 Dijkstrin algoritam

Dijkstrin algoritam rješava problem najkraćeg puta u grafu s nenegativnim težinama bridova, pri čemu graf ne mora biti nužno usmjeren. Počevši s početnim vrhom s kao korijenom, Dijkstrin alg. pravi stablo najkraćeg puta koje sadrži najkraće putove od vrha s do svih ostalih vrhova grafa. Tijekom procesa vrh grafa je nedohvaćen, dohvaćen ili obrađen. Vrh koji već pripada stablu je obrađen. Ukoliko je vrh u obrađen, tada je pronađen najkraći put P^* od vrha s do vrha u i njegova duljina je poznata: $u.d = \omega(P^*)$. Vrh koji je susjedan obrađenom vrhu je dohvaćen. Primjetimo da je posjećeni vrh također dohvaćen. Ukoliko je vrh u dohvaćen, put P od s do u ne mora biti nužno najkraći. Vrh u koji nije dohvaćen je nedohvaćen i za taj vrh vrijedi $u.d = \infty$. Algoritam posjećuje vrhove na osnovu njihove udaljenosti od ishodišnog vrha.

Sam algoritam se bazira na apstraktnoj strukturi podataka - redovima s prioritetima u kojima svaki element ima "prioritet" koji mu omogućuje prednost pred ostalim elementima. Zapravo, možemo reći da redovi s prioritetima pohranjuju skup disjunktnih elemenata i da je svakom elementu pridružen ključ koji predstavlja njegovu prioritetnu vrijednost. Redovi s prioritetima se mogu implementirati na više načina. Jednostavna implementacija bi bila pomoću povezanih listi, ali ne i efikasna što se tiče vremenske složenosti određenih operacija. Za daljne potrebe razumijevanja postavimo da je n maksimalan broj elemenata reda. Dodavanje elemenata(engl. Insert) u povezanu listu odgovara $O(1)$ vremenskoj složenosti. Međutim, DecreaseKey operacija koja "smanjuje" vrijednost ključa (mijenja vrijednost ključa s nekom manjom vrijednosti) i operacija ExtractMin, koja briše minimalni element iz reda te ga vraća, nas koštaju $\theta(n)$ vremena. Detaljnije o prioritetnim redovima se može pogledati u [1] .

Dijkstrin Algoritam

Dijkstra(G,ω,s)

1. *za svaki vrh* $v \in G.V$ *radi sljedeće:*
2. $v.d = \infty$
3. $v.\pi$=*NIL*
4. $s.d$=*0*
5. S=$\emptyset$
6. Q=$G.V$
7. *dok je* $Q \neq \emptyset$ *radi sljedeće:*
8. *u=ExtractMin(Q)*
9. $S = S \cup \{u\}$
10. *za svaki vrh* $v \in G.Adj[u]$ *radi sljedeće:*
11. *Relax(u,v,ω)*

Relax(u,v,ω)

1. *ako je* $v.d > u.d + \omega(u,v)$ *radi sljedeće:*
2. $v.d = u.d + \omega(u,v)$

3. $v.\pi = u$

Algoritam radi na sljedeći način: u početnom dijelu postavlja udaljenost svih vrhova na privremene udaljenosti koje iznose neki vrlo velik broj odnosno u ovom zapisu - beskonačno. Prethodnik svakog vrha se postavlja na NIL vrijednost. Nadalje, udaljenost ishodišnog vrha s(od vrha s) se postavlja na nulu, pravimo prazan skup S i skup Q koji će sadržavati sve vrhove tj. udaljenosti zadanog grafa G. Prva, *while* petlja radi sve dok je Q neprazan skup. U koraku 8 koristimo funkciju ExtractMin koja iz skupa, odosno reda implementiranog pomoću binarne hrpe, briše i vraća minimalni element - minimalnu udaljenost. Prilikom prvog prolaska kroz *while* petlju vrijedi da je $u = s$. Petlja for u koraku 10 prolazi kroz sve susjedne vrhove vrha u i vrši relaksaciju brida (u, v). Funkcija $Relax(u, v, \omega)$ ažurira privremene vrijednosti udaljenosti svakog susjednog vrha, zapravo ona ih "smanjuje" na način da im pridružuje vrijednost $min = \{v.d, u.d + \omega(u, v)\}$. Kažemo da Dijkstrin algoritam vrši relaksaciju bridova (u, v). Lako je za primjetiti da će se *while* petlja u koraku 7 izvršiti $|V|$ puta, jer će se prilikom svakog novog izvršavanja ove petlje ukloniti jedan vrh iz skupa Q. Dijkstra uvijek bira najbliži vrh v tj. vrh sa najmanjom vrijednosti $v.d$ u skupu $S \setminus V$. Ukoliko tražimo najkraći put od ishodišnog do ciljnog vrha, algoritmu se može dodati uvjet zaustavljanja koji će zaustaviti pretraživanje kada dođemo do cilja.

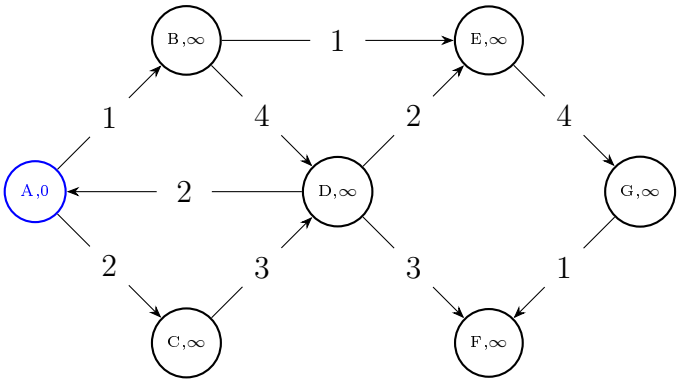
Da bismo ispisali pronađeni najkraći put od s do t poslužit ćemo se informacijama koje smo skupili o prethodnicima svakog vrha na najkraćem putu i funkcijom $Ispis(G, s, t)$.

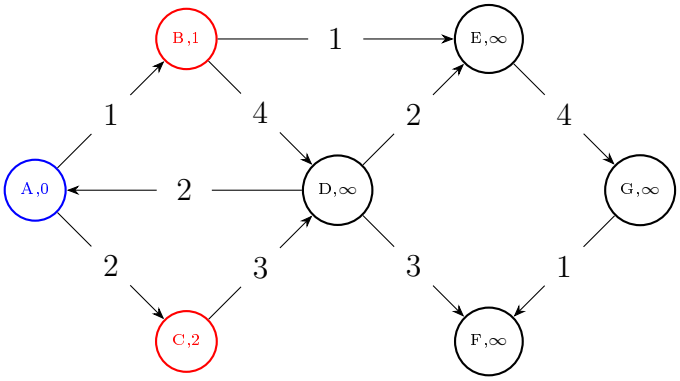
Ispis(G,s,t)

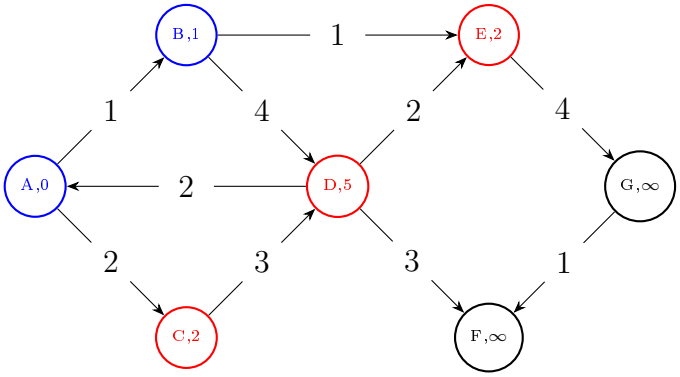
1. *ako je s=t ispiši t, inače radi sljedeće:*
2. *ako je* $\pi(t) = NIL$ *ispiši:"Ne postoji put od s do t" inače radi sljedeće:*
3. *Ispis(G,s,*$\pi(t)$*),ispiši t*

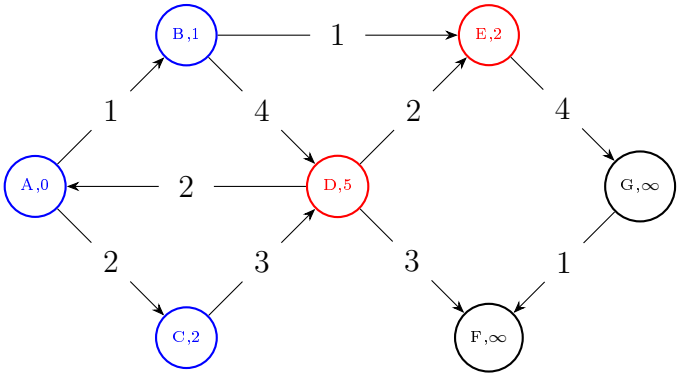
Funkcija rekurzivno prolazi putem od t do s pomoću sačuvanih prethodnika π i na povratku ispisuje svaki vrh na putu od s do t.

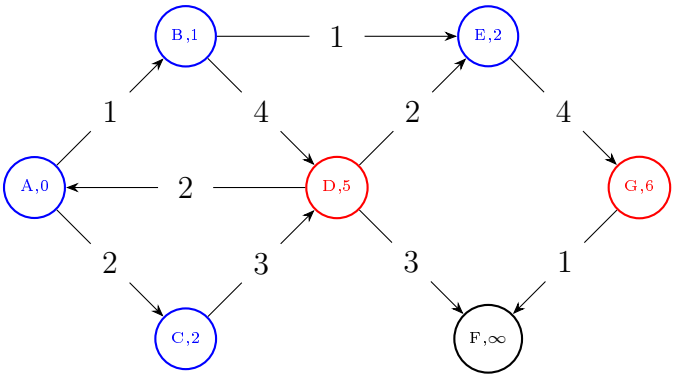
Primjer 2.2. *Primjer Dijkstrina algoritma korak po korak na danom grafu, pri čemu je dan početni vrh A. Računaju se najkraći putovi do svih vrhova.*

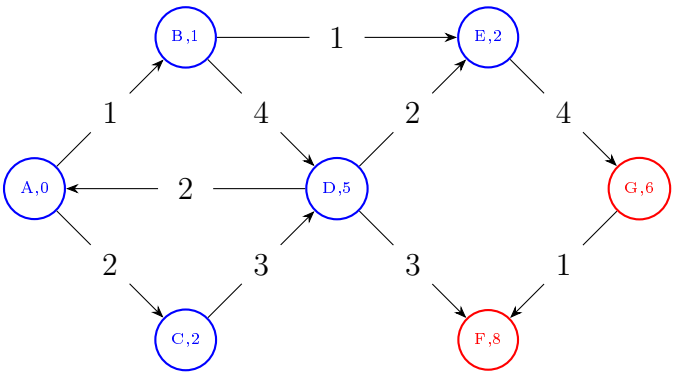

B,∞
E,∞
A,0
D,∞
G,∞
C,∞
F,∞
1
2
3
4

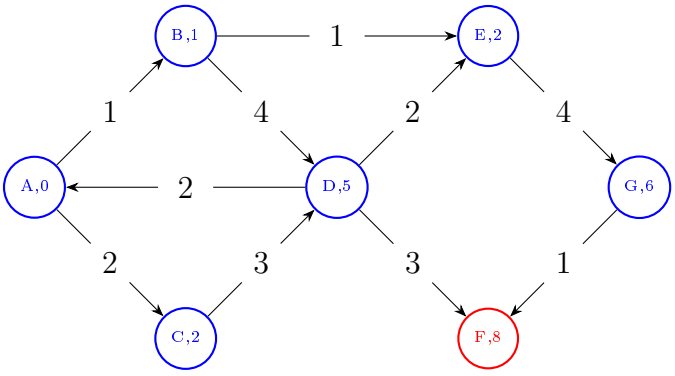

B,1
E,∞
A,0
D,∞
G,∞
C,2
F,∞
1
2
3
4

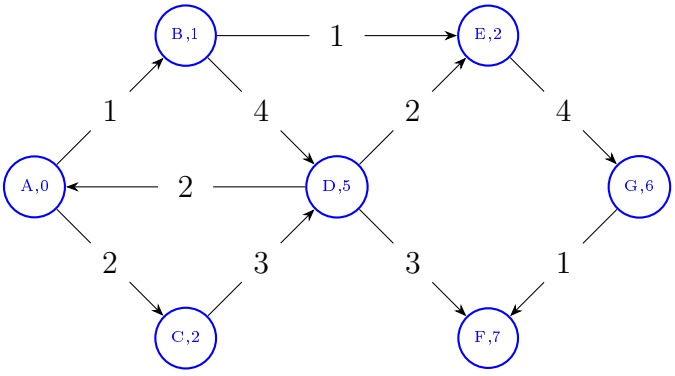

B,1
E,2
A,0
D,5
G,∞
C,2
F,∞
1
2
3
4


B,1
E,2
A,0
D,5
G,∞
C,2
F,∞
1
2
3
4









Najkraći putovi od vrha A do svih ostalih vrhova su:
$A \to B$
$A \to C$
$A \to B \to D$ ili $A \to C \to D$
$A \to B \to E$

$A \to B \to E \to G$
$A \to B \to E \to G \to F$

Zbog toga što u svakom koraku uzima vrh s najkraćom udaljenosti, Dijkstrin algoritam spada u pohlepne algoritme. Pohlepna strategija u Dijkstrinom algoritmu osigurava dolaženje do optimalnog rješenja.

Teorem 2.1. *[1](Ispravnost Dijkstrina algoritma)*
Dijkstrin algoritam, primijenjen na težinskom, usmjerenom grafu $G = (V, E)$ s nenegativnom težinskom funkcijom ω i početnom vrhu s, završava s $u.d = \delta(s, u), \forall u \in V$ odnosno Dijkstra točno izračunava sve najkraće putove.

2.4.1 Brzina Dijkstrina algoritma

Algoritam sadrži minimalni red s prioritetima Q koji koristi tri operacije: Insert (umetanje elemenata), ExtractMin i DecreaseKey (koristimo za relaksaciju). Operacije Insert i ExtractMin se pozivaju jednom za svaki vrh. Svaki vrh $u \in V$ je dodan jednom u skup S i zato je svaki brid u listi susjedstva pregledan jednom u *for* petlji. Ukupan broj bridova je $|E|$ te se prema tome *for* petlja izvrši $|E|$ puta i zato se operacija DecreaseKey poziva najviše $|E|$ puta.

Vrijeme izvršavanja algoritma ovisi o načinu implementacije redova. Korištenjem polja to vrijeme iznosi $O(V^2)$. Algoritam možemo poboljšati korištenjem binarnih hrpa. ExtractMin koristi vrijeme $O(\log_2 V)$, a takvih operacija je $|V|$. Vrijeme potrebno da se stvori minimalna binarna hrpa iznosi $O(V)$. Vrijeme izvršavanja DecreaseKey operacije je $O(\log_2 V)$, a takvih je operacija ukupno $|E|$. Tada je ukupno vrijeme izvršavanja $O(V \log_2 V + E \log_2 V)$.

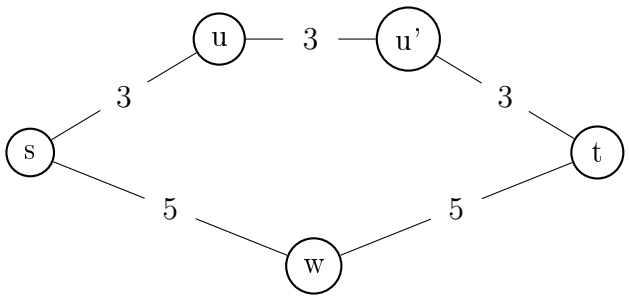
U daljnjem tekstu objasnit ćemo dvije tehnike ubrzanja Dijkstrin algoritma: dvosjmerni Dijkstrin algoritam i A*.

2.4.2 Dvosmjerni Dijkstrin algoritam

Dvosmjerni Dijkstrin algoritam također rješava problem najkraćeg puta od vrha s do vrha t. Osnovna ideja dvosmjernog pretraživanja je naizmjenično pretraživanje grafa pomoću *pretraživanja unaprijed* i *pretraživanja unazad*. Počinjemo pretraživanjem unaprijed koje koristi klasičan Dijkstrin algoritam pri čemu je početni vrh s. Nakon prvog koraka stajemo te slijedi pretraživanje unazad od vrha t. Potrebno je napomenuti da pretraživanje unazad prati bridove grafa u obratnom smjeru. Kao što smo postavljali vrijednost udaljenosti svakog vrha na ∞, osim početnog, sličnom tehnikom ćemo se koristiti i kod pretraživanja unazad, osim što će sada svi vrhovi imati vrijednost ∞, a vrh t vrijednost 0. Prema tome, imat ćemo dva prioriteta koja će odgovarati pripadajućim pretraživanjima. Ovaj način pretraživanja zahtijeva dupliranje struktura podataka jer sve što imamo za jedan smjer, moramo imati i za obratni smjer. Pojavljuju se dva reda Q_f i Q_b za svako pretražvanje.

Postavlja se pitanje, kada stati s pretraživanjem? S pretraživanjem stajemo kada neki vrh u bude obrađen u oba smjera pretraživanja, tj. kada vrh bude obrisan iz Q_f i Q_b. Neka π_f normalno prati bridove, a π_b prati bridove unazad. Ukoliko je vrh u prvi obrisan iz oba reda, najkraći put pronalazimo koristeći π_f od s do u i π_b od t do u. Ovaj algoritam nije baš potpuno točan, tj. vrh u ne mora biti na najkraćem putu. Pogledajmo primjer kada to ne mora vrijediti.

Primjer 2.3. *[4] Primjenom dvosmjernog pretraživanja dobili bismo da je duljina najkraćeg puta od vrh s do vrha t 10, jer bi prvi obostrano posjećeni vrh bio w, što pokazuje da moramo obaviti određena preračunavanja kako bismo došli do stvarnog najkraćeg puta. Kako bi to ispravili moramo pronaći vrh v koji ima minimalnu vrijednost $d_f(v) + d_b(v)$.*

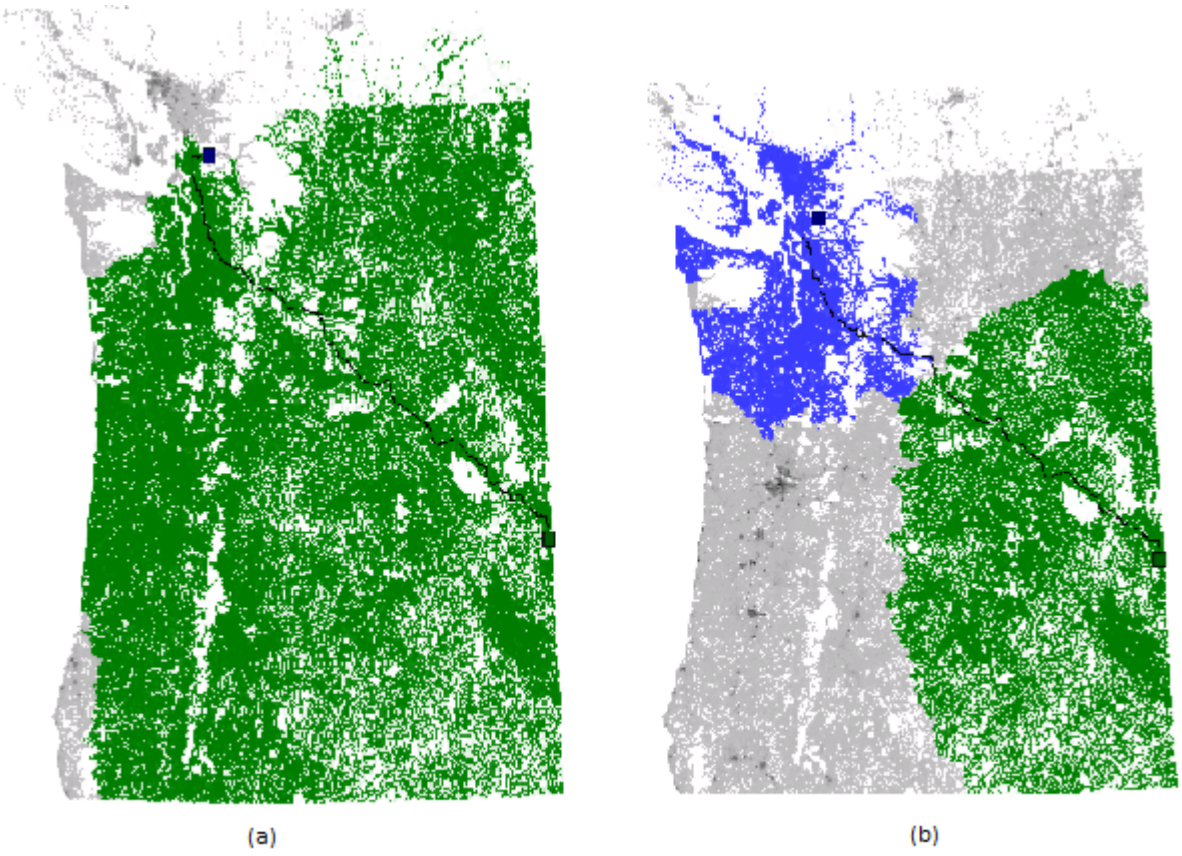


$$d_f(u) + d_b(u) = 3 + 6 = 9$$

$$d_f(u') + d_b(u') = 6 + 3 = 9$$

$$d_f(w) + d_b(w) = 5 + 5 = 10$$

Kako bi uspješno izbjegli ovakav scenarij koristimo jači kriterij zaustavljanja. Moramo održavati udaljenost μ trenutno najboljeg puta. U početku je $\mu = \infty$. Nakon što je ispitan brid (v, u) prilikom pretraživanja unaprijed i ispitan vrh u prilikom pretraživanja unazad, ažuriramo μ ukoliko je $d_f(v) + \omega(v, u) + d_b(u) < \mu$. Sličan postupak vrijedi i za suprotan smjer pretraživanja. Neka su top_f i top_b najviše vrijednosti hrpe (unaprijed i unazad). Tada se dvosmjerni Dijkstrin algoritam zaustavlja kada je $top_f + top_b \geq \mu$.
Zašto ovo vrijedi? Pretpostavimo da postoji $s - t$ put P duljine manje od μ. Tada mora postojati brid (v, u) na tom putu P tako da je: $\delta(s, v) < top_f$ i $\delta(u, t) < top_b$. Nakon što su v i u provjereni tada je pronađen i P. To je kontradikcija da takav P postoji.



Slika 5: (a) širenje Dijkstra algoritma ,(b) širenje dvosmjernog Dijkstra algoritma

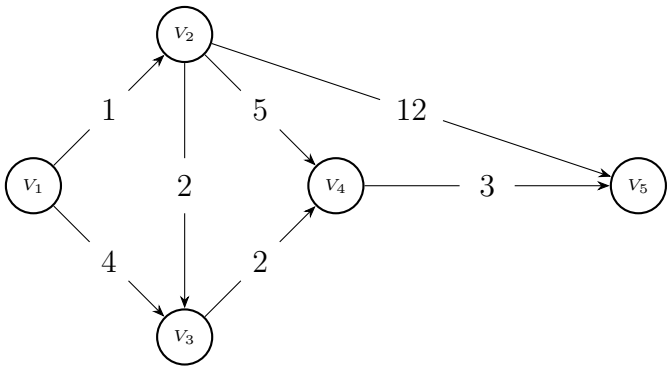
2.4.3 Usmjereno pretraživanje ili A*

A* pretraživanje, tehnika iz polja umjetne inteligencije, je usmjereno pretraživanje tj. proces dobija "osjećaj za smjer". Za svaki vrh potrebno je postaviti donju granicu s obzirom na udaljenost ciljnog vrha. Algoritam A* obično smanjuje broj vrhova grafa koje treba ispitati. U svakom koraku procesa vrh v se bira na način da minimizira privremenu udaljenost od početnog vrha s plus donja granica udaljenosti do ciljnog vrha t. Učinak A* pretraživanja ovisi o dobrom odabiru donjih granica. Ukoliko vrhovi grafa predstavljaju geografske koordinate i ako tražimo najkraće (ne i najbrži) put, tada se Euklidska udaljenost može koristiti kao donja granica. Sada slično Dijkstrinu algoritmu prilikom svake iteracije glavne petlje A* ispituje donju granicu $f(u) = g(u) + h(u)$ gdje je $g(u)$ duljina puta od početnog vrha do vrha u, a $h(u)$ je procjena tj. algoritam koristi heuristiku koja procjenjuje udaljenost do ciljnog vrha. Dijsktra je poseban slučaj A* pretraživanja uz uvjet da je heuristika 0.

Uz A* možemo spomenuti i ALT algoritam koji koristi sličnu ideju. ALT (A* search, landmarks, triangle inequality) algoritam je baziran na A* pretraživanju, orijentirima i nejednakosti trokuta. Nakon odabira malog broja orijentira, za svaki vrh v izračunamo udaljenost $d(v, l)$ i $d(l, v)$ za svaki orijentir l. Za vrhove v i t nejednakost trokuta određuje dvije donje granice $d(l, t) - d(l, v) \leq d(v, t)$ i $d(v, l) - d(t, l) \leq d(v, t)$. Maksimum ovih donjih granica se upotrebljava prilikom A* pretraživanja. Npr. koristeći 16 orijentira dovoljno je da se

postigne faktor ubrzanja oko 27 u Zapadnoj Europi gdje se cestovna mreža sastoji od oko 18 miliona vrhova. Međutim, metoda orijentira zahtijeva dosta memorije budući da preračunava dvije udaljenosti za svaki par vrh-orijentir. Stoga, zaključak je da ALT pretraživanje ima optimalno ubrzanje, ali koristi previše memorije za velike cestovne mreže odnosno grafove.

Primjer 2.4. *Primijenimo sada sve tri metode prilikom pronalaženja najkraćeg puta između vrhova V_1 i V_5. Za postavljanje vrijednosti heuristike h koristili smo Euklidsku udaljenost.*$d[v] = [d[v_1], d[v_2], d[v_3], d[v_4], d[v_5]]$



vrh	vrijednost heuristike(h)
V_1	7
V_2	6
V_3	2
V_4	1
V_5	0

	d[v]	Dijkstra	dvosm. Dijkstra	A *
0		[0,∞, ∞, ∞, ∞]	[0,∞, ∞, ∞, ∞],[∞, ∞, ∞, ∞,0]	[0,∞, ∞, ∞, ∞]
1		[**0**,1,4,∞, ∞]	[**0**,1,4,∞, ∞],[∞,12,∞,3,**0**]	[**7**,7,6,∞, ∞]
2		[**0**,**1**,3,6,13]	[**0**,**1**,3,6,13],[∞,8,5,**3**,**0**]	[**7**,7,**6**,7,∞]
3		[**0**,**1**,**3**,5,13]	[**0**,**1**,**3**,5,13],[9,7,**5**,**3**,**0**]	[**7**,**7**,**5**,7,13]
4		[**0**,**1**,**3**,**5**,8]	-	[**7**,**7**,**5**,**6**,13]
5		[**0**,**1**,**3**,**5**,**8**]	-	[**7**,**7**,**5**,**6**,**8**]

3 Dodatak

Prilažemo Python implementaciju kao demonstraciju računanja optimalne rute ne Euklidskom grafu G gdje vrhove čine 20-tak glavnih gradova u HR. Program za uneseno polazište i uneseno odredište računa optimalnu rutu u G.

```
import googlemaps
import sys
import heapq




class Vrh:
    def __init__(self, vrh):
        self.id = vrh
        self.susjedi = {}
        self.posjecen = False
        self.udaljenost = sys.maxsize
        self.prethodnik = None

    def __lt__(self, other):
        if isinstance(other, self.__class__):
            return self.udaljenost < other.udaljenost
        return NotImplemented

    def dodaj_susjeda(self, susjed, tezina=0):
        self.susjedi[susjed] = tezina

    def postavi_udaljenost(self, udaljenost):
        self.udaljenost = udaljenost

    def postavi_prethodni(self, prethodnik):
        self.prethodnik = prethodnik

    def postavi_posjecen(self):
        self.posjecen = True

    def dohvati_susjede(self):
        return self.susjedi.keys()

    def dohvati_tezinu(self, susjed):
        return self.susjedi[susjed]

    def dohvati_udaljenost(self):
        return self.udaljenost

    def dohvati_id(self):
        return self.id




class Graf:
    def __init__(self):
        self.br_vrhova = 0
        self.vrhovi = {}

    def __iter__(self):
        return iter(self.vrhovi.values())

    def dodaj_vrh(self, cvor):
        self.br_vrhova = self.br_vrhova + 1
        novi_vrh = Vrh(cvor)
```

```
59          self.vrhovi[cvor] = novi_vrh
60          return novi_vrh
61
62      def dodaj_brid(self, v, u, tezina = 0):
63          if v not in self.vrhovi:
64              self.dodaj_vrh(v)
65          if u not in self.vrhovi:
66              self.dodaj_vrh(u)
67
68          self.vrhovi[v].dodaj_susjeda(self.vrhovi[u], tezina)
69          self.vrhovi[u].dodaj_susjeda(self.vrhovi[v], tezina)
70
71      def postavi_prethodni(self, v):
72          self.prethodnik = v
73
74      def dohvati_vrh(self, v):
75          if v in self.vrhovi:
76              return self.vrhovi[v]
77          else:
78              return None
79
80      def dohvati_vrhove(self):
81          return self.susjedi.keys()
82
83      def dohvati_prethodni(self, v):
84          return self.prethodnik
85
86
87
88  def dijkstra(graf, s, t):
89
90      s.postavi_udaljenost(0)
91      red = [(v.dohvati_udaljenost(),v) for v in graf]
92      heapq.heapify(red)
93
94      while len(red)!=0:
95          u = heapq.heappop(red)
96          v = u[1]
97          v.postavi_posjecen()
98          if t==v:
99              break
100         else:
101             for n in v.susjedi:
102                 if n.posjecen:
103                     continue
104                 udaljenost = v.dohvati_udaljenost() \
105                              + v.dohvati_tezinu(n)
106
107                 if udaljenost < n.dohvati_udaljenost():
108                     n.postavi_udaljenost(udaljenost)
109                     n.postavi_prethodni(v)
110
111                 heapq.heapify(red)
112
113
114
115 def najkraci_put(v, put):
116     if v.prethodnik:
```

```
117             put.append(v.prethodnik.dohvati_id())
118             najkraci_put(v.prethodnik, put)
119         return
120 
121 
122 
123 def convert_str(input_str):
124 
125     if not input_str and not isinstance(input_str, str):
126         return 0
127     out_number = ''
128     for ele in input_str:
129         if (ele == '.' and '.' not in out_number) or ele.isdigit():
130             out_number += ele
131         elif out_number:
132             break
133     return float(out_number)
134 
135 
136 
137 if __name__ == '__main__':
138 
139 
140     print('obrada podataka,molim Vas pricekajte...')
141     gmaps=googlemaps.Client(key='API key')
142 
143     h = Graf()
144 
145     Gradovi=[('osijek','vukovar'),('vukovar','vinkovci'),\
146              ('osijek','slatina,Hrvatska'),\
147              ('slatina,hrvatska','virovitica'),\
148              ('vinkovci','zupanja'),\
149              ('zupanja','slavonski brod'),\
150              ('slavonski brod','djakovo'),('osijek','djakovo'),\
151              ('djakovo','vinkovci'),\
152              ('djakovo','pozega,hrvatska'),\
153              ('pozega,hrvatska','sisak'),\
154              ('virovitica','bjelovar'),\
155              ('bjelovar','koprivnica'),('bjelovar','krizevci'),\
156              ('bjelovar','zagreb'),('zagreb','krizevci'),\
157              ('krizevci','koprivnica'),('varazdin','koprivnica'),\
158              ('zagreb','varazdin'),('sisak','karlovac'),\
159              ('karlovac','zagreb'),('karlovac','rijeka'),\
160              ('rijeka','porec'),('rijeka','rovinj'),\
161              ('rijeka','pula'),('rijeka','zadar'),\
162              ('zadar','karlovac'),('karlovac','knin'),\
163              ('knin','sibenik'),('zadar','knin'),\
164              ('sibenik','knin'),('knin','sinj'),\
165              ('sibenik','split'),('split','makarska'),\
166              ('sinj','makarska'),('makarska','metkovic'),\
167              ('makarska','dubrovnik'),('metkovic','dubrovnik'),\
168              ('djakovo','bjelovar'),\
169              ('slavonski brod','pozega,hrvatska'),\
170              ('pozega,hrvatska','bjelovar'),('zagreb','sisak')]
171 
172 
173     for i in range(0,len(Gradovi)):
174         if Gradovi[i][0]  not in h.vrhovi:
```

```
175             h.dodaj_vrh(Gradovi[i][0])
176
177         if Gradovi[i][1] not in h.vrhovi:
178             h.dodaj_vrh(Gradovi[i][1])
179
180
181
182     for i in range(0,len(Gradovi)):
183         h.dodaj_brid(Gradovi[i][0],Gradovi[i][1],\
184                         convert_str(gmaps.distance_matrix
185                         (Gradovi[i][0],Gradovi[i][1])\
186                         ['rows'][0]['elements'][0]['distance']['text']))
187
188     a=input('Pocetak:')
189     b=input('Cilj:')
190     a=a.lower()
191     b=b.lower()
192
193
194     lista_gradova=list(h.vrhovi.keys())
195     if a  not in lista_gradova or b not in lista_gradova:
196         print('Trazeni gradovi nisu dostupni.
197         Pokusajte s gradovima: %s'
198         %(list(h.vrhovi.keys())))
199
200
201
202     dijkstra(h,h.dohvati_vrh(a),h.dohvati_vrh(b))
203
204     t = h.dohvati_vrh(b)
205     put = [t.dohvati_id()]
206
207     najkraci_put(t, put)
208     print ('Najkraci put od grada %s do grada %s
209     je prolazeci kroz gradove : %s.Duljina puta iznosi %.2f km'
210     %(a,b,put[::-1],(h.dohvati_vrh(b)).dohvati_udaljenost()))
```

Literatura

[1] T.H. Cormen,C.E.Leiserson,R.L.Rivest,C.Stein, *Introduction to Algorithms,3rd Edition*, MIT Press,2009.

[2] T.Harju, *Lecture Notes on Graph Theory*, Department of Mathematics University of Turku, 2011.

[3] R.Diestel, *Graph theory*,2000.

[4] Peter Sanders,Dominik Schultes, *Engineering Fast Route Planning Algorithms*

[5] Dominik Schultes, *Route Planning in Road Networks*,Karlsruhe Institute of Technology,2008.